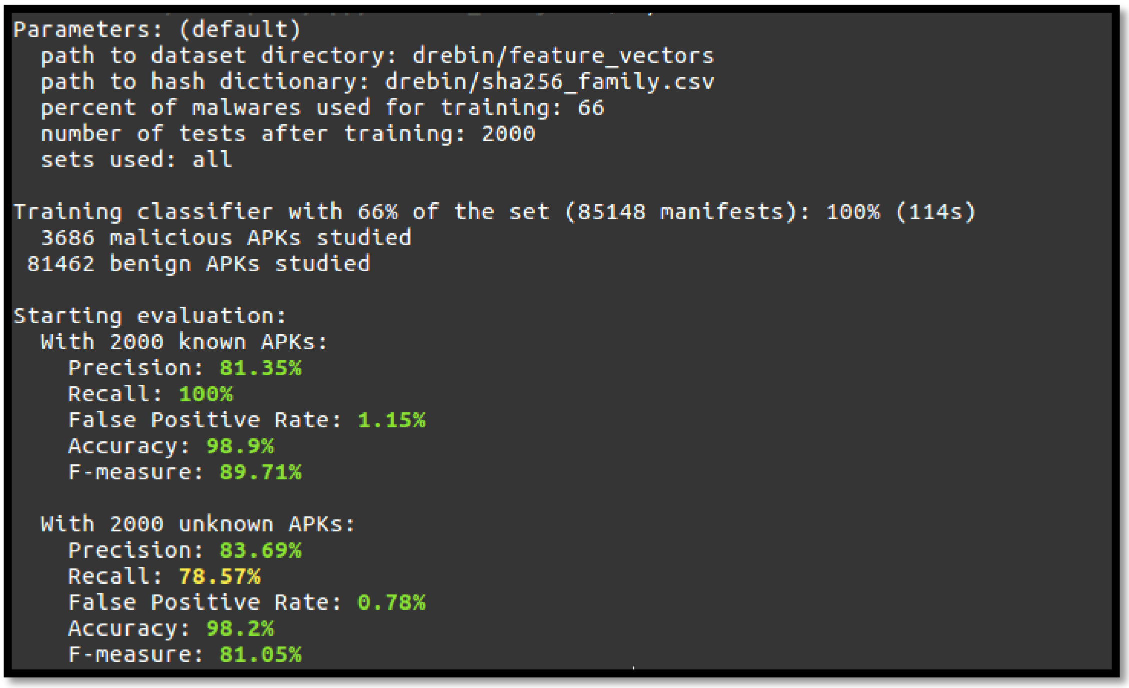
Aurélien BEC
aurelien.bec@gmail.com
1786704

November the 19th

Machine Learning
Homework I

Malware analysis for machine learning



```
Parameters: (default)
  path to dataset directory: drebin/feature_vectors
  path to hash dictionary: drebin/sha256_family.csv
  percent of malwares used for training: 66
  number of tests after training: 2000
  sets used: all

Training classifier with 66% of the set (85148 manifests): 100% (114s)
  3686 malicious APKs studied
 81462 benign APKs studied

Starting evaluation:
  With 2000 known APKs:
    Precision: 81.35%
    Recall: 100%
    False Positive Rate: 1.15%
    Accuracy: 98.9%
    F-measure: 89.71%

  With 2000 unknown APKs:
    Precision: 83.69%
    Recall: 78.57%
    False Positive Rate: 0.78%
    Accuracy: 98.2%
    F-measure: 81.05%
```

Problem

The problem here is to develop two programs able to classify some android app's manifests, telling if the application is a malware or not. And additional work is to learn to recognize the family of a malware.

We are using the Drebin dataset to perform the reading of the manifests.

The work detailed in the next sections contains some information on the malware detector and the family classifier.

The whole project can be found on GitHub, following this link:
https://github.com/AurelBec/ML-malware_analysis

Important information:

- To compile the project only using the .zip and not the GitHub, type the following commands:

```
$ g++ -o detector *.cpp -std=c++0x -O2 -I -libboost -lboost_system -lboost_filesystem
$ g++ -o classifier *.cpp -std=c++0x -O2 -I -libboost -lboost_system -lboost_filesystem
```

Only the name of the output changes, but it is important in the main function. The function uses the first argument (the name) to decide if we want to use the detector or the classifier. That's why the cpp files included are the same and why a bad-named program won't work.

- If boost is not installed, use this command to solve the issue:

```
$ sudo apt-get install libboost-all-dev
```

- The first execution of the program usually takes a long time, but after, it's more quickly compute, within less than 2 minutes.

Method and Algorithm

To address the problem, I chose to use the naive Bayes approach. It's easy to implement and to understand. Moreover, I realized the second exercise about the spam filter using this method, and it allowed me to reuse some parts.

Naive Bayes classifier

This method easy to understand, we start by studying a known dataset, and knowing the number of times a feature is or not in the different classes, we can guess the class an unknown example belongs.

Bayes theorem is mathematically expressed as:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Training

For our problem, we can set A to the probability that the application is malware and B as the contents of the manifest.
If P(A|B) > P(¬A|B) then we can classify the application as malware, otherwise we can't.
Note that since Bayes' Theorem results in a divisor of P(B) in both cases, we can remove it from the equation for our comparison. This leaves: P(A)*P(B|A) > P(¬A)*P(B|¬A).
Calculating P(A) and P(¬A) is trivial, they are simply the percentage of your training set which are malware versus not malware.

The idea is to read all the content of the different files, and create of graph for each of the eight subsets. Each line of the manifest is composed of two parts. The prefix gives us the subset the feature belongs. Then we add the feature to the graph. To do this, I separate the feature in little value, using separators such as ",./-"… each of this little value obtain are the nodes of the graph. At the end of a branch is kept the number of times the sequence appears. If a value is not known, we simply create a new branch.

For example, this is the result reading a malware manifest and printing the graph:

```
Class Malware (1 studied)
 SubSet Hardware components
   android hardware touchscreen 1
   android hardware telephony 1

 SubSet Requested permissions
   android permission read phone state 1
   android permission receive sms 1
   android permission receive boot completed 1
   android permission internet 1
   android permission install packages 1
   android permission vibrate 1
   android permission send sms 1
   android permission restart packages 1
   android permission write external storage 1

 SubSet App components
   proinactivity 1
   com uniplugin sender areceiver 1
   com convertoman proin fservices 1
   fservices 1
```

```
SubSet Filtered intents
  android intent action boot completed 1
  android intent action main 1
  android intent action phone state 1
  android intent category launcher 1

SubSet Restricted API calls
  java net httpurlconnection 1
  android telephony telephonymanager getdeviceid 1
  android content context startactivity 1
  android app notificationmanager notify 1

SubSet Used permissions
  android permission vibrate 1
  android permission read phone state 1
  android permission internet 1

SubSet Suspicious API calls
  getdeviceid 1
  printstacktrace 1
  getsystemservice 1

SubSet Network addresses
  http yandex ru 1
  http rukodelniza ru phoneconvert commander php 1
  http rukodelniza ru phoneconvert otstuk php 1
  yandex ru 1
  rukodelniza ru 1
```

For the first subset, the root as one child "android" which has only one too, "hardware", but this last has two children, "touchscreen" and "telephony", each one, with an occurrence of 1.
I also put all I read in lower case, to compute the same way HTTP and http for example.

Classifying

To decide if a random manifest belongs to a malware or not, we need to compute the value $p(C|F_1,\dots,F_n)$.
Doing this this for all the classes, and keeping the higher value, we estimate the right class.
This probability is equals to:

$$p(C|F_1,\dots,F_n) = \frac{p(C)\ p(F_1,\dots,F_n|C)}{p(F_1,\dots,F_n)}.$$

And

$$p(F_1,\dots,F_n|C) = p(F_1|C)\ p(F_2|C)\ p(F_3|C)\ \cdots\ p(F_n|C) = \prod_{i=1}^{n} p(F_i|C).$$

The value F_i is the occurrence of the features, at the end of the branch. The probability is this value divides by the number of features known in the class.
P(C) is the number of examples of this class we studied, divides by the total number of examples.
As said before, we don't need to compute $p(F_1, \dots, F_n)$

A trick I used is to perform operations in the log space. This avoid little number multiplications.

Programs

The two programs are identical. The only difference is in the main function, which will check if the running command is “./detector” or “./classifier”

Using one or the other will create an instance of MalwareDetector or FamillyClassifier, both inherited of NaiveBaysClassifier.

Those two classes redefines only two methods: Train() and Evaluate().
As we don’t consider the same dataset in the both cases, we need to adapt the method building the graphs.

Parameters

Different parameters can be used to run the two programs:
Those information can be obtained by typing “./xxxxx help”

```
Binary Classifier for android applications
 Outputs: MALWARE | NON MALWARE
 Method: Bayesian approach
 Options:
  --percent=NUMBER
    Defines the percentage of the set to use for training (0 - 100)
    default: 66
  --nbtests=NUMBER
    Defines the number of tests to run for evaluating
    default: 2000
  --sets=NUMBER, ..., NUMBER
    Defines the sets used by the classifier
    default: ALL
  --pathDS=PATH
    Defines the path to the directory containing the manifests
    default: drebin/feature_vectors
  --pathHD=PATH
    Defines the path to the file containing the SHA1 HASH of known malwares
    default: drebin/sha256_family.csv

Aurelien BEC, nov. 2017
```

```
Family Malware Classifier for android applications
 Outputs: FAMILY
 Method: Bayesian approach
 Options:
  --percent=NUMBER
    Defines the percentage of the set to use for training (0 - 100)
    default: 66
  --nbtests=NUMBER
    Defines the number of tests to run for evaluating
    default: 2000
  --sets=NUMBER, ..., NUMBER
    Defines the sets used by the classifier
    default: ALL
  --pathDS=PATH
    Defines the path to the directory containing the manifests
    default: drebin/feature_vectors
  --pathHD=PATH
    Defines the path to the file containing the SHA1 HASH of known malwares
    default: drebin/sha256_family.csv

Aurelien BEC, nov. 2017
```

Results

Malware detector

It’s obvious than our classifier must have a higher accuracy than 50% percent, the accuracy of a random classifier, but also over 94%. This value is the one given by classifier always saying that the application is not a malware. By chance, we are better:

```
Parameters: (default)
  path to dataset directory: drebin/feature_vectors
  path to hash dictionary: drebin/sha256_family.csv
  percent of malwares used for training: 66
  number of tests after training: 2000
  sets used: all

Training classifier with 66% of the set (85148 manifests): 100% (113s)
  3686 malicious APKs studied
 81462 benign APKs studied

Starting evaluation:
  With 2000 known APKs:
    Precision: 77.27%
    Recall: 100%
    False Positive Rate: 1.3%
    Accuracy: 98.75%
    F-measure: 87.17%

  With 2000 unknown APKs:
    Precision: 83.33%
    Recall: 74.32%
    False Positive Rate: 0.57%
    Accuracy: 98.5%
    F-measure: 78.57%
```

Using different parameters:

10% of the dataset for training:

```
[…]
  percent of malwares used for training: 10

[…]
Starting evaluation:
  With 2000 known APKs:
    Precision: 90.29%
    Recall: 100%
    False Positive Rate: 0.52%
    Accuracy: 99.5%
    F-measure: 94.89%

  With 2000 unknown APKs:
    Precision: 95.91%
    Recall: 54.65%
    False Positive Rate: 0.1%
    Accuracy: 97.95%
    F-measure: 69.62%
```

Only the sets 2, 3 and 8:

```
[…]
  sets used: S2-Requested permissions
             S3-App components
             S8-Network addresses
[…]
Starting evaluation:
  With 2000 known APKs:
    Precision: 81.05%
    Recall: 98.71%
    False Positive Rate: 0.93%
    Accuracy: 99.05%
    F-measure: 89.01%

  With 2000 unknown APKs:
    Precision: 81.17%
    Recall: 73.4%
    False Positive Rate: 0.83%
    Accuracy: 97.95%
    F-measure: 77.09%
```

Family classifier

The results given by the family classifier are not as complete as those displayed by the detector. I didn't find a way to compute anything expect accuracy, which is the number of correct guesses divide by the number of tries.

Running the program with the default parameters give us this result:

```
Parameters: (default)
  path to dataset directory: drebin/feature_vectors
  path to hash dictionary: drebin/sha256_family.csv
  percent of malwares used for training: 66
  number of tests after training: 2000
  sets used: all

Training classifier with 66% of the set (3669 malwares): 100% (0s)
   22 malware families considered (n > 20)

Starting evaluation:
  With 2000 known APKs:
    Accuracy: 99.61%

  Warning: Not enough unknown APKs, will use 1891 instead of 2000
  With 1891 unknown APKs:
    Accuracy: 72.63%
```

We can notice the warning message, it's just a security saying that there are only 1891 unknown manifests in the unknown set and therefore, running over 2000 examples is impossible.
The classifier works well, returning the good family in less than 3 times in 4.

With other parameters, we have the following results:

With 10% of the dataset

```
[…]
  percent of malwares used for training: 10

[…]
Training classifier with 10% of the set (556 malwares): 100% (0s)
   6 malware families considered (n > 20)

Starting evaluation:
  Warning: Not enough known APKs, will use 556 instead of 2000
  With 556 known APKs:
    Accuracy: 99.6%

  With 2000 unknown APKs:
    Accuracy: 57.86%
```

With sets 2,3 and 8

```
[…]
  sets used: S2-Requested permissions
             S3-App components
             S8-Network addresses

Training classifier with 66% of the set (3669 malwares): 100% (0s)
   22 malware families considered (n > 20)

Starting evaluation:
  With 2000 known APKs:
    Accuracy: 99.53%

  Warning: Not enough unknown APKs, will use 1891 instead of 2000
  With 1891 unknown APKs:
    Accuracy: 71.35%
```